Memory Forensics of a Java Card Dump

Jean-Louis Lanet, Guillaume Bouffard, Rokia Lamrani, Ranim Chakra, Afef Mestiri, Mohammed Monsif, Abdellatif Fandi

Memory Forensics of a Java Card Dump

Jean-Louis Lanet[1], Guillaume Bouffard[2], Rokia Lamrani[3], Ranim Chakra[3], Afef Mestiri[3], Mohammed Monsif[3], and Abdellatif Fandi[3]

[1] INRIA, LHS PEC
263 Avenue Général Leclerc, 35042 Rennes,
jean-louis.lanet@inria.fr
http://secinfo.msi.unilim.fr/lanet/
[2] Agence Nationale de la Sécurité des Systèmes d'Informations,
51, boulevard de La Tour-Maubourg, 75700 Paris 07 SP, France.
guillaume.bouffard@ssi.gouv.fr
[3] University of Limoges, 123 Avenue Albert Thomas
87060 Limoges, France

Abstract. Nowadays several papers have shown the ability to dump the EEPROM area of several Java Cards leading to the disclosure of already loaded applet and data structure of the card. Such a reverse engineering process is costly and prone to errors. Currently there are no tools available to help the process. We propose here an approach to find in the raw data obtained after a dump, the area containing the code and the data. Then, once the code area has been identified, we propose to rebuilt the original binary CAP file in order to be able to obtain the source code of the applet stored in the card.

Keywords: Java Card, Memory Forensics, Reverse Engineering, Disassembler, Index of coincidence

1 Introduction

Several attacks have been successful in dumping the memory of the smart card and in particular the EEPROM. Even if the cards are more protected nowadays, it is still possible to get the memory contents. Then, to reverse the content of the dump, one must analyze kilobytes of raw data for obtaining the expected information. At present, there are no tools available for reversing the memory dump for a Java based smart card. The EEPROM memory is made of data for the system and the applications, and their metadata (data descriptors), encrypted data (secure key container), Java CAP file and in particular Java byte code and sometimes native code. For a reverse engineer it is a hard task to find the adequate information and the tools used for reversing a Java Card memory dump are missing. So we have developed a disassembler which is based on natural language recognition and heuristics. Each binary language has a signature that takes into account the probability of occurrence of the language elements. Thus each card embeds at least two languages with two different signatures. Then, for

the Java part, a symbolic execution of the recognized program verifies the Java type system for increasing the confidence in the recognition probability. A pattern recognition phase which is card dependent, is then performed to recognize the metadata stored in the card. Having a precise knowledge of the software and in particular the Control Flow Graph (CFG) could be helpful for new attacks or understanding the algorithms.

The rest of the paper is organized as follows: the second section presents the security model of the card and the state of the art concerning the attacks in order to obtain a memory dump. The third section introduces the dump file analyzer and its index of coincidence for recognizing different machine languages. The fourth section explains the Java Card Disassembler and Analyzer (JCDA) implementation followed by experimentation results. Future works and conclusion end this paper..

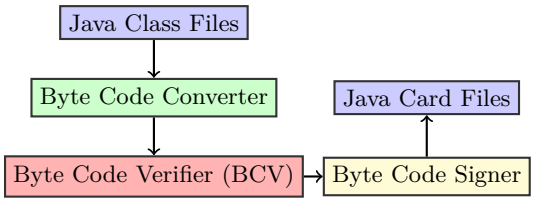
2 Java Card

The Java platform [12] has been downsized for fitting the smart card constraints. The Java Card technology offers a multi-application environment where sensitive data must be protected against illegal access from another applet. The classical Java technology uses three security elements - type verification, class loader and security managers to protect each applet. Embedding the Java security elements into a smart card is not possible due to the resource constraints.These components have been adapted to the specific requirements of Java Card.

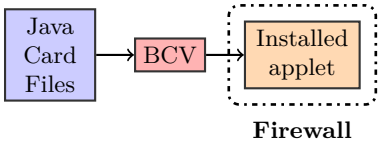
2.1 Security in the Java Card World

To be compliant with the Java security rules, the Java Card security model is split in two parts. One, outside the card (Fig. 1(a)) is in charge of preparing the code to be loaded into the card. It includes a Byte Code Verifier (BCV), a converter and a mechanism to ensure integrity and/or confidentiality of the code to be loaded. The BCV is in charge of verifying the semantics of the Java-program. It ensures that the checked applet file format respects the specification (structural verification) and that all methods are well formed and verify the type system of Java. It is a complex process involving an elaborate program analysis using a very costly algorithm in terms of time consumption and memory usage. Next is the Java Card converter which translates each Java Card package into a Java Card-Cap. A Java Card-Cap is a lightweight Java Card-Class based on the tokens. This file format is designed to be optimized for the resource-constraint devices. The organization which provides the applet must sign[4] the application for the on-card loader that will verify the signature. This verification ensures the loader the origin of the code, and thus that the code is compliant with the Java security rules.

[4] Due to security reasons, the ability to download code into the card is controlled by a protocol defined by Global Platform [15]. This protocol ensures that the owner of the code has the necessary authorization to perform the action.



(a) *Off-card* security model.



(b) *On-card* security model.

Fig. 1: The Java Card Security Model.

The second part of the security model is embedded into the smart card (Fig. 1(b)). The loader verifies the signature and optionally a BCV might verify the Java security compliance of the Java-CAP file to be installed. Currently, just a few Java Cards embed an on-card BCV component. The applet to be installed is linked after some potential checks. Once an applet is installed, the segregation of different applets is enforced by the firewall which is based on the package structure of Java Card and the notion of context.

2.2 Attacks on Java Card

Recently, the idea to inject physical fault to bypass the BCV's checks has emerged. A legitimate applet which complies with the Java Card security rules is installed into a Java Card. With the help of a fault injection, an attacker can modify some memory content which can lead to exploitable deviant behavior. So the application mutates to execute a malicious byte code which can break the security model. Classically, the fault attacks are used to attack cryptographic algorithm implementations [1, 8, 14].

Barbu et al. [3] succeed to bypass the embedded smart card BCV. In order to perform it, a correct applet is installed which contains an unauthorized cast between two different objects. Statically, the applet is compliant with the Java Card security rules. If a laser beam hits the bus in such a way that the cast type check instruction is not executed, this applet becomes hostile and can execute any shell code. This type of attack exploits a new method to execute illegal instructions where the physical and logical levels are perturbed. This method succeeds only on some cards and others seem to not be sensitive to this attack.

Bouffard et al. [4], proposed a way to perturb the applet's CFG with a laser beam injection into the smart card's non-volatile memory. The authors described

the attack on a loop `for`, but it can be extended with other conditional instructions.

The Java Card specification [12] defines two instructions to branch at the end of a loop, a `goto` and the `goto_w` instructions. The first one branches with a 1-byte offset and the second one takes 2-byte offset. Since the smart card's memory manager stores the array data after the memory byte code, a laser fault on the high part of the `goto_w` parameter can shift the backward jump to a forward one and the authors succeeded to execute the contents of an array. Unlike Barbu et al., Bouffard et al. described a persistent attack to execute their shellcode. Hamadouche et al. [7], proposed a way to obtain Java Card API addresses embedded in the card. With this attack it is possible to use the Java Card internal references to execute a rich shellcode.

Lancia [11] explained an exploitation on Java Card instance allocation based on high precision fault injection. Instead of the Java Virtual machine, each instance created by the Java Card Runtime Environment (JCRE) is allocated in a persistent memory[5]. On the modern Java Card Memory Management Unit (MMU), references are represented by an indirect memory address. This address is an index to a memory address pool which in turn refers to a global instance pool managed by the virtual machine. Like a persistent type confusion, Lancia presented an attack of the global instance pool mechanism in which a laser beam shifts the index referred in the byte code. During the applets execution, the JCRE resolves an index with the associated address in the global instance pool table and have access to another instance of the expected object. This persistent modification may offer information of the card as a part of the smart card memory.

Each attack previously described here gives information about the targeted Java Card. These information can contain a part or the whole Java Card memory (essentially the EEPROM part) as raw data. Generally, this memory fragment contains program's code (virtual and/or native) and data (system and application) needed to well-execute each installed applet. These raw data are called the *dump file*.

Since this work is done without any knowledge of the features implemented in the card, it's of a prime importance to be able to recognize the different elements, to separate code and data. Until now, carving the smart card memory dump is done manually. It s a difficult task, prone to human errors and long. To automate this analysis, we propose a Java Card disassembler to reverse the Java Card memory.

2.3 State of the Art of Memory Carving

Memory analysis is an important part of effective computer forensics. There has been significant research done in improving analysis of memory dump files [17,

[5] The Java Card specification [12] provides some functions to create transient objects. The data of the transient object stored in the RAM memory, but the header of this object is always stored in the persistent memory.

18]. Forensic memory analysis starts with collecting the memory from the target machine followed by parsing the memory dump into meaningful artifacts. The technique consists of several steps: parsing the internal memory structures, retrieving the assembly code and stack from the memory, constructing the control flow graph from the executable code and reversing it and finally identifying the data structures. Unfortunately, these techniques rely on well-known characteristics of the operating system. Furthermore, in most cases, these tools only work on a small number of operating system versions. The absence of a general methodology for forensic log analysis has resulted in ad-hoc analysis techniques such as log analysis [13] and operating system-specific analysis [5].

Schuster [17] proposes an approach to define signatures for executive object structures in the memory and recover the hidden and lost structures by scanning the memory looking for predefined signatures. However, defining a signature that uniquely identifies most of the data structures is not achievable except for a small set of kernel structures.

Walters et al. [18] present an approach for extracting in-memory cryptographic keying material. They have presented an extensible framework which is able to automatically derive object definitions from C source code and extract the underlying objects from memory.

A particular effort has been done for retrieving information from volatile memory that might determine if encryption is being used and extract volatile artifacts and passwords/passphrases [10]. Their approach considers that access to and acquisition of live encrypted data requires that these data must be in the clear in order to be manipulated. Since the contents of an encrypted container or volume are available to the user, then if physical access is gained to the live machine while it is in this state, the contents will also be accessible.

A disassembler recognizes the code section by parsing the whole memory and building the CFG. It recognizes the end of the code by a return instruction and cancels the current analysis if a byte does not represent an instruction.

Most of works on memory carving try to extract data from the dump of general purpose operating system. Of course with such systems, data and code are separated and they proceed by pattern matching for retrieving the data. In our tool, we need first to recognize the virtual and native code and then to recognize the data. Techniques usually used in recognizing code cannot be applied here since some instructions of the code are undocumented.

3 Memory Carving on Java Card

3.1 A Memory Dump

A *dump file* contains a set of binary values which represents a fragment of the smart card memory. The program's code and data can be found in the smart card memory and these information are sensitive.

In the Listing 1.1, a fragment of a Java Card memory dump is presented. The targeted smart card embeds a Java Card 2.2.1 and Global Platform 2.1.1 with

36 kB of EEPROM, 128 kB of ROM and 2 kB of RAM. This dump corresponds to an 88-byte fragment of the EEPROM and starts from the logical address `0x13f8`.

Listing 1.1: A fragment of a Java Card memory dump.

```
0x13f0:                          00 0b 81 00 0a 48 65 6c
0x1400: 6c 6f 57 6f 72 6c 64 00 00 02 80 00 00 03 04 02
0x1410: 0c 34 00 00 01 be 81 08 00 0a 00 19 00 25 00 01
0x1420: 2e 00 01 0d 48 65 6c 6c 6f 57 6f 72 6c 65 41 70
0x1430: 70 01 71 00 02 34 04 00 04 06 02 00 00 01 73 01
0x1440: 75 00 05 42 18 8d 08 97 18 01 87 06 18 01 87 07
0x1450: 18 08 91 00 07 87 08 18 01 87 09 1e 29 04 03 29
```

A reversed version of the dump is listed in the Listing 1.2 after intensive work, the first part of the analyzed dump contains metadata (package and class) information. The second part describes the byte code method of a class.

Listing 1.2: The reverse of the value listed in the Listing 1.1

```
0x13f8: 000b 81 00 // Array header: data size: 0x000b,
                   //               type: 0x81, owner: 0)
              0a 48 65 6c 6c 6f 57 6f 72 6c 64 // PACKAGE_AID
0x1408: 00 0002 8000 0003 0402 0c34 0000 // Unknown data
0x1414: 01be 81 08 // Array header: data size: 0x01be,
                   //               type: 0x81, owner: 08)
        000a 0019 0025 0001 2e00 010d    // Undefined values
        48 65 6c 6c 6f 57 6f 72 6c 64 41 70 70 // APPLET_AID
        01 71 00 02 34 04 00 04 06 02 00 00 01 73 01 75 00
0x1442:   /* method 00: */
          // Method's header
          05 // flags: 0 max_stack: 5
          42 // nargs: 4 max_locals: 2
          // Method's bytecode
          18       // aload_0
          8d 08 97 // invokestatic 0x0897
          18       // aload_0
          01       // aconst_null
          87 06    // putfield_a 06
// To be continued ...
```

In the above sample, the byte codes used are the same as that defined in the Java Card specification [12]. Depending on the card, the program code may be scrambled [2, 16] or new and undocumented instructions used. In the previous case, the card *xor*s the value of each instruction to mask the code. But even with any masking or encryption, the stored program in the memory always keep the same semantics. In the last case, we have no information about the semantics of the code which can be compressed. For that reason it is impossible to use a simple execution to find the methods. The only way is to use an approximative approach.

3.2 Index of Coincidence

In 1922, Friedman [6] invented the notion of Index of Coincidence (IC) to reverse ciphered message. In cryptography, this technique consists of counting number of times the identical letters appear in the same position in both texts. This count can be calculated either as a ratio of the total or normalized divided by the expected count for a random source model. The IC is computed as defined in the equation 1.

$$\mathrm{IC} = \frac{\sum_{i=1}^{c} n_i(n_i - 1)}{N(N-1)/c} \tag{1}$$

where N is the length of the analyzed text and n_i is the frequencies of the c letters of the alphabet (c=26 for a Latin alphabet).

The IC is mainly used both in the analysis of natural-language text and in the analysis of ciphered message (as cryptanalysis). Even when only a ciphered message is available, the coincidences in the ciphered text can be caused by coincidences in the plain message. For example, this cryptanalysis technique is mainly used to attack the Vigenère cipher. The IC for a natural-language like French language is 0.0778, English is 0.0667 and German is 0.0762.

3.3 Finding Java Card Byte Codes

In a Java Card memory *dump file*, it is very difficult to separate the program's data and code. The program's byte code can be assimilated to a language where each instruction has a precise location in the language's grammar.

A Java Card byte code operation is composed by an instruction (between the range `0x00` to `0xB8`) and potentially a set of bytes as argument. The Java Card toolchain ensures that the built Java Card byte codes are in compliance with the rules of Java language. The Friedman's approach is mainly based on the analysis of a whole cyphered text. In our case, a *dump file* includes data, byte codes and random values. Random values are a set of bytes which represent old system's values partially overridden or no longer used by the system. To find where a method's byte codes is located, we decided to compute the Friedman's equation upon a sliding window. To determine the IC value for the Java Card byte codes, we tested a set of Java Card byte code built by the Oracle's toolchain. An acceptable IC for Java Card byte codes is located between 0.020 and 0.060.

Computing the IC value upon the sliding window is equivalent to perform the equation 1 with each byte inside the interval. With different sizes of the sliding window, IC value is computed. The results are presented in the Fig. 2.

On this figure, the method's area to discover is located between the vertical dashed lines. We show that the optimal size for the sliding windows is between 135 and 150. This range includes false positive closed to the method's area. It

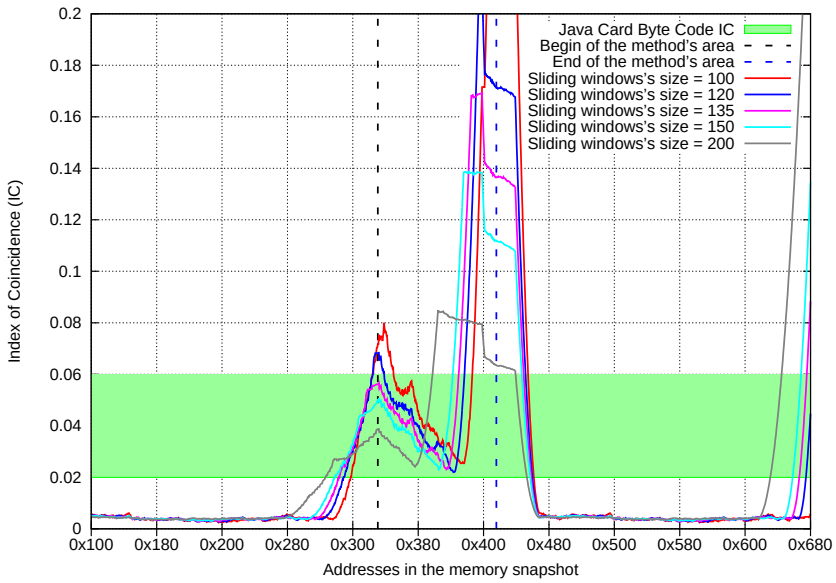


Fig. 2: Searching the optimal sliding window's size.

is due to the size of the sliding window which includes a part of the method's byte code. False positive are detected by heuristics. We used several heuristics to eradicate the false positives. A code should not embed the value `0x00` which corresponds to the `nop` opcode except for operands. The size of the operands cannot exceed two bytes (except the specific case of the `switch`). The program decides that three consecutive bytes having the value zero cannot represent code. Another heuristic concerns the undefined byte code, above a given level of such bytes the program cancel the current window.

3.4 Finding Data in a Java Card Memory Dump

Carving raw data requires to characterize the objects manipulated by the system. Any data stored in the smart card belonging to the Java Card world contains a header (or metadata) which describes the data type, data owner and, sometimes, the size of the data. Inside a Java Card, the data can be:

– a package information. The package AID and all the classes in the card should be saved to verify the ownership context during the execution of an applet.

– a class information. A class contains the initialization value of each field, the method's byte code and an AID. A fragment is presented with its package information, in the Listing 1.2.
– the instance of a class which refers to each field sets to the current value (regarding to the life cycle of the instance). The instance of the class is linked with an instance AID which can be different from that of the class AID.
– an array is the last element which contains a header. We discovered empirically that an array header includes the size of the array data, the data type and the owner's context. An example of Java Card array found in a memory dump is shown in the Listing 1.3.

Listing 1.3: A Java Card array found in the memory dump

```
0010  // Data size
81    // type of the data, there it is a byte array
08    // applet's owner context
/* Data */
CA FE CA FE CA FE CA FE CA FE CA FE CA FE CA FE
```

4 JCDA: Java Card Disassembler and Analyzer

To implement the memory carving approach for Java Card, we have developed a tool, named Java Card Disassembler and Analyzer (JCDA) written in Java which aims to reverse a Java Card memory dump. It has been designed to be adapted for the architecture of each Java Card.

To reverse a Java Card memory dump, the JCDA (Fig. 3) requires a card model and a *dump file*. The first one defines the structure of the data contained in the smart card memory dump. This model is a high level abstraction of how the smart card stores objects and associated instances, array, etc. in the memory. This file should be filled by the attacker. This step has not yet been automated, it needs to create in the card objects of different nature (arrays, instances,...) and to compare the state of the memory before and after the creation of the object. The second parameter is a *dump file* of an attacked Java Card.

In our tool, reversing a Java Card memory dump is split in two steps. In the first step, the Java Card Analyzer searches in the Java Card memory *dump file* to locate the Java code or native code. As described previously, to find the Java Card instruction, an automatic process based on the index of coincidence is performed. The card memory model is used to search information about the classes, arrays and other data by using pattern matching.

The second step in the JCDA starts with the disassembling of the Java code recognized in the previous step. Its aim is to reverse each applet installed in the memory dump. The idea is to rebuild a partially stored CAP file in the *dump file*. Once rebuilt, Oracle provides a tool to convert a CAP file to a CLASS file. Then, it becomes obvious to convert the CLASS to Java file, many tools exist for

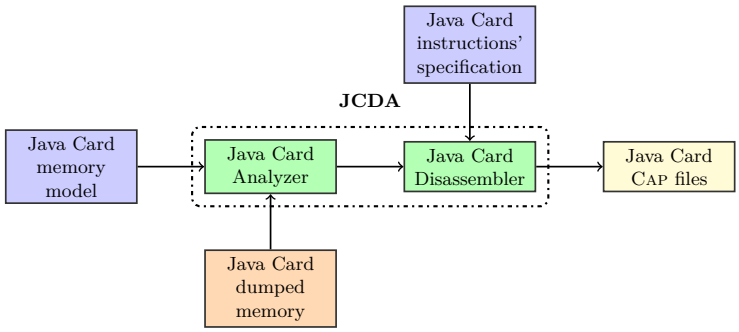


Fig. 3: JCDA Architecture.

that purpose. In this whole process, the main difficulty consists in regenerating CAP file from an applet installed into the memory. It implies some constraints:

1. Due to software and hardware limitations, information not needed to execute the applet are deleted during the installation file. To restore the complete CAP file, the dependencies between each component are used. For instance, the `Directory` and the `Descriptor` components, often not kept after the installation step, are generated using the information contained in the `Class`, `Method`, and `Static Field` components. Regarding to the smart card's installer implementation, our prototype needs to know how the card stores each CAP file component in memory;
2. In a CAP file, the `Import` component refers each class, method, and field imported by the application. Generating this component from a linked and installed applet rises some problems. In fact, each tokens is linked by the smart card internal references during the loading step. To reverse it, a learning step, based on the Hamadouche et al.'s attack [7], which maps the smart card Application Programming Interface (API) is needed. This map links smart card's internal references and the function's name. With the function's name, we are able to rebuild the `Import` component upon the EXPORT files.
3. Finally, the generated CAP file shall be validated by the BCV. This step is mandatory to translate the CAP to CLASS file.

Moreover, the constraints defined by the CAP file dependencies imply to respect a precise order to generate the CAP file components, as illustrated in the Fig. 4. Introduced by this figure, the method component is the keystone of our approach. Indeed, each `Method` contains references to its own class, its class's fields, etc.

For the analyzer, the methods area is a byte stream. To process the CAP file dependencies, the disassembler should find each method present in the dumped area before starting the reverse. This step aims to split the byte stream into a set of method. From the first byte of the method's area, an in-depth analysis is done to detect the end of each method. This analysis is based on the abstract execution

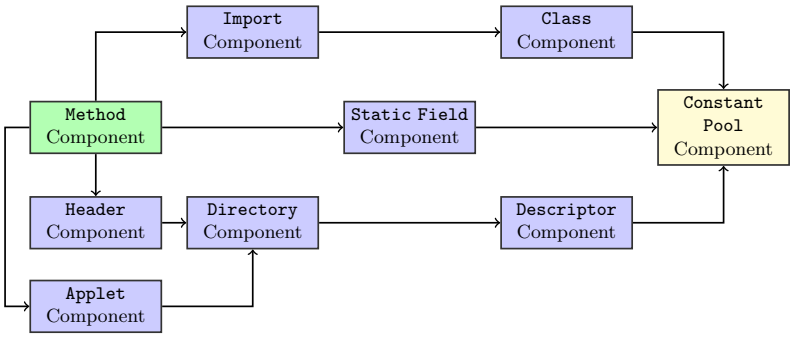


Fig. 4: Order of generation of CAP file components.

of each instruction. In the case of an unknown instruction, the analyzer warns the user for this occurrence. This instruction has be added to the set of specific instructions for this card model. For example, one smart card manufacturer has replaced the `invokestatic` byte code by an undocumented one.

5 Experimental Results

On a smart card snapshot memory, we succeed in detecting the applets byte codes with the IC approach. The analyzer gave information shown in the Listing 1.4 to the disassembler.

Listing 1.4: Linked applet from a memory snapshot.

```
4868 6666 6666 6666 6666 4868 6666 6666 6666 6666 4170 7000
2c00 5480 0102 3400 ff00 0408 0002 0056 ffff 004e 0069 0057
0059 005c 005e 0049 004a 0223 0408 090a 0b04 0062 0019 007e
0009 0062 8019 0085 0000 0083 8007 007d 0000 0092 8002 0075
0000 0110 188d 0897 188b 0101 7a02 308f ffac 3dcc ffee 3b7a
0110 0478 0010 7a00 107a 0010 7a01 4004 7801 1010 4278 0110
0478 0010 7a03 2319 8b01 012d 188b 0103 6003 7a10 0681 181a
1007 8118 1c10 0881 181c 1a03 10ca 381a 0410 fe38 1199 998d
08c6 701c 2e11 6789 8d08 c611 9999 8d08 c670 0d28 0411 9999
8d08 c615 0493 7a00 04c1 06ff ffff ff00 08c2 06aa aaaa aaaa
aaaa aa00 04c3 0601 0001
```

Regarding to the card model, this snapshotted area is parsed as presented in the annex A, Listing 1.8. First, we filled, from the dump, some fields inside the `Method`, `Class`, `Header`, `Applet` and the `Static Field` component. Due to the limited size, the initial vector of the static fields is not kept in the card memory. This information is also lost for us. To regenerate the static fields initialization vector, we decided to use the current values another option would be the Java default value. This is one of the limits of the approach.

The next step aims to build the `Constant Pool` component. From the CAP file, this component have purposed to describe the type of each token used by the application. Tokens are used in the `Method`, `Class`, `Static Field` and `Descriptor` components.

In the `Method` component, each instruction with a reference as argument was linked by an internal reference during the CAP file installation. Converting each internal reference creates the set of token used by the application and aims to regenerate the `Import` component. There, the API mapping is used to describe the token to the correct EXPORT file. Once `Import` component is regenerated, we have enough information to create the `Descriptor` and `Constant Pool` components.The Listing 1.5 exhibits a fragment of entries in `Constant Pool` component restored from the dump.

Listing 1.5: Rebuilt `Constant Pool` component: internal tokens

```
/*0000, 0*/ CONSTANT_ClassRef : 0x0001 // first class
/* offset class constructor 0x0 => method_info[10] (@21) */
/*0004, 1*/ CONSTANT_StaticMethodRef : 0x0021
/*0008, 2*/ CONSTANT_StaticFieldRef  : 0x0006
/*000c, 3*/ CONSTANT_StaticFieldRef  : 0x0008
```

To find the external tokens description, we need to link the references in the *dump file* with the card API to obtain information of each token in the `Constant Pool`. Then we replace them with an index incremented at each occurrence, Listing 1.6.

Listing 1.6: Rebuilt `Constant Pool` component: external tokens

```
            // applet's constructor
/*0010, 4*/ CONSTANT_StaticMethodRef : 0x81,0x3,0x1
            // register function's token
/*0014, 5*/ CONSTANT_VirtualMethodRef: 0x81,0x3,0x1
            // APDU.getbuffer function's token
/*0018, 6*/ CONSTANT_VirtualMethodRef: 0x81,0xa,0x1
            // selectingApplet function's token
/*001c, 7*/ CONSTANT_VirtualMethodRef: 0x81,0x3,0x3
            // ISOException.throwIt method's token
/*0020, 8*/ CONSTANT_StaticMethodRef : 0x81,0x7,0x1
            // Exception class' token
/*0024, 9*/ CONSTANT_ClassRef        : 0x80,0x2
```

Finally, when each other components have been regenerated, the `Directory` component is built.

To validate globally the approach, we checked the regenerated CAP file with the BCV. As shown in the Listing 1.7, our generated file as the correct structure and it contains coherent values regarding the Java Card specification. For this proof of concept we did not regenerate the Java source file but this is not an issue.

Listing 1.7: Analyzing of regenerated CAP file by the Oracle BCV.

```
[INFO:] Verifier [v3.0.4]
[INFO:]    Copyright (c) 2011, Oracle and/or its affiliates.
           All rights reserved.
[INFO:] Verifying CAP file dumpedCapFile.cap
[INFO:] Verification completed with 0 warnings and 0 errors.
```

6 Future works and Conclusions

We have developed a proof of concept of a tool-chain that allows to recover from raw data application code and objects. In order to find Java code we used the index of coincidence to detect any byte code area. With a pattern matching algorithm, we are able to recover instances in the memory. The JCDA is still an ongoing academic development and currently only few card memory models can be recognized. We only focus for the moment to the byte code language and we need further development for the native language. This will be very useful for the improvement of JCDA development. A second improvement concerns the ability to automate the pattern learning phase for the card model, which is currently a manual process. We only recognize basic objects (array, AID,...) another improvement should be to recognize specific instances like secure containers for key storage. A last improvement for our work is integrating our tool into the IDA Disassembler [9]. IDA is a software which implements all the features required to reverse a computer application. This software is mainly used by security laboratories. One intrinsic limit concerns the initialization vector of the fields for which the information is lost after the applet installation. For example, if the value of a PIN code is stored into a static array its value will never be recoverable which is a good point from the security point of view. As soon as we have a stable version we expect to provide it as open source project for the academic community.

References

1. Aumüller, C., Bier, P., Hofreiter, P., Fischer, W., Seifert, J.P.: Fault attacks on RSA with CRT: Concrete Results and Practical Countermeasures. IACR Cryptology ePrint Archive 2002, 73 (2002)
2. Barbu, G.: On the security of Java Card platforms against hardware attacks. Ph.D. thesis, TÉLÉCOM ParisTech (2012)
3. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on java card 3.0 combining fault and logical attacks. In: Gollmann, D., Lanet, J.L., Iguchi-Cartigny, J. (eds.) CARDIS. Lecture Notes in Computer Science, vol. 6035, pp. 148–163. Springer (2010)
4. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.L.: Combined software and hardware attacks on the java card control flow. In: Prouff, E. (ed.) CARDIS. Lecture Notes in Computer Science, vol. 7079, pp. 283–296. Springer (2011)

5. Dolan-Gavitt, B.: Forensic analysis of the windows registry in memory. Digital Investigation 5, 26–32 (2008)
6. Friedman, W.F.: The index of coincidence and its applications in cryptography. Aegean Park Press (1922)
7. Hamadouche, S., Bouffard, G., Lanet, J.L., Dorsemaine, B., Nouhant, B., Magloire, A., Reygnaud, A.: Subverting byte code linker service to characterize java card api. In: Seventh Conference on Network and Information Systems Security (SAR-SSI). pp. 75–81 (May 22rd to 25th 2012), https://sarssi2012.greyc.fr/
8. Hemme, L.: A Differential Fault Attack Against Early Rounds of (Triple-)DES. In: Joye, M., Quisquater, J.J. (eds.) CHES. Lecture Notes in Computer Science, vol. 3156, pp. 254–267. Springer (2004)
9. Hex Rays, S.: IDA Pro Disassembler and Debugger
10. Klein, T.: All your private keys are belong to us. Tech. rep., trapkit (Feb 2006)
11. Lancia, J.: Java card combined attacks with localization-agnostic fault injection. In: Mangard, S. (ed.) CARDIS. Lecture Notes in Computer Science, vol. 7771, pp. 31–45. Springer (2012)
12. Oracle: Java Card 3 Platform, Virtual Machine Specification, Classic Edition 3.0.0. Oracle (Sep 2011)
13. Peikari, C., Chuvakin, A.: Security warrior - know your enemy. O'Reilly (2004)
14. Piret, G., Quisquater, J.J.: A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD. In: Walter, C.D., Çetin Kaya Koç, Paar, C. (eds.) CHES. Lecture Notes in Computer Science, vol. 2779, pp. 77–88. Springer (2003)
15. Platform, G.: Card Specification v2.2. March (2006)
16. Razafindralambo, T., Bouffard, G., Thampi, B.N., Lanet, J.L.: A dynamic syntax interpretation for java based smart card to mitigate logical attacks. In: Thampi, S.M., Zomaya, A.Y., Strufe, T., Calero, J.M.A., Thomas, T. (eds.) SNDS. Communications in Computer and Information Science, vol. 335, pp. 185–194. Springer, Trivandrum, India (2012)
17. Schuster, A.: Searching for processes and threads in Microsoft Windows memory dumps. Digital Investigation 3(Supplement-1), 10–16 (2006)
18. Walters, A., Petroni, N.: Integrating volatile memory forensics into the digital investigation process. Blackhat Hat DC (2007)

A Content of a dumped area

Listing 1.8: The content of the dump file used for test.

```
48 68 66 66 66 66 66 66 66 66 // PACKAGE AID
/* APPLET Component */
48 68 66 66 66 66 66 66 66 66 41 70 70 // Applet AID
002c //
0054 // @Method Install
// Class Component
// interface_info
80  // -> flag
01  // -> interface_count

// Class_info
```

```
0234 // -> super_class_ref
00 // -> declared_instance_size
ff // -> first_reference_index
00 // -> reference_count
04 // -> public_method_table_base
08 // -> public_method_table_count
00 // -> package_method_table_base
02 // -> package_method_table_count
0056 ffff 004e 0069 0057 0059 005c 005e // public_methods

0049 004a // package_methods

// Implemented interface info
0223 // class_ref interface
04 // -> count
[ 08 09 0a 0b ] // index

/* Method component */
0400 6200 1900 7e00 0900 6280 1900 8500
0000 8380 0700 7d00 0000 9280 0200 7500
0001 1018 8d08 9718 8b01 017a 0230 8fff
ac3d ccff ee3b 7a01 1004 7800 107a 0010
7a00 107a 0140 0478 0110 1042 7801 1004
7800 107a 0323 198b 0101 2d18 8b01 0360
037a 1006 8118 1a10 0781 181c 1008 8118
1c1a 0310 ca38 1a04 10fe 3811 9999 8d08
c670 1c2e 1167 898d 08c6 1199 998d 08c6
700d 2804 1199 998d 08c6 1504 937a

/* Static Field Component */
0004 c1 06 ff ff ff ff          // byte array
0008 c2 06 aaaa aaaa aaaa aaaa // short array
0004 c3 06 01 00 01 00          // Boolean array
```